Technical Report NEESgrid-2003-03
www.neesgrid.org

Draft Whitepaper Version: 1.0
Last modified: January 10, 2003

Protocol Specification for the NSDS, Driver and DAQ

Paul Hubbard[1]

[1] Argonne National Laboratory

Feedback on this document should be directed to hubbard@mcs.anl.gov

Introduction

This document is for people wishing to understand the NSDS-driver-DAQ interactions, either for debugging or in the interest of replacing the driver or DAQ code. Sites customizing the DAQ code should also find this of use. Other documentation is planned for those incorporating this code into their sites.

It is assumed that the reader is familiar with the NEESgrid project and its goals. If not, information can be found at http://www.neesgrid.org/.

Terminology

NSDS means NEESgrid Streaming Data Server. It's a piece of Java-based code that handles security, client subscriptions, streaming data out to the world, and so forth.

DAQ means data acquisition system. The NEESgrid reference platform is LabVIEW from National Instruments. We provide example code in CVS that implements the functionality in this document.

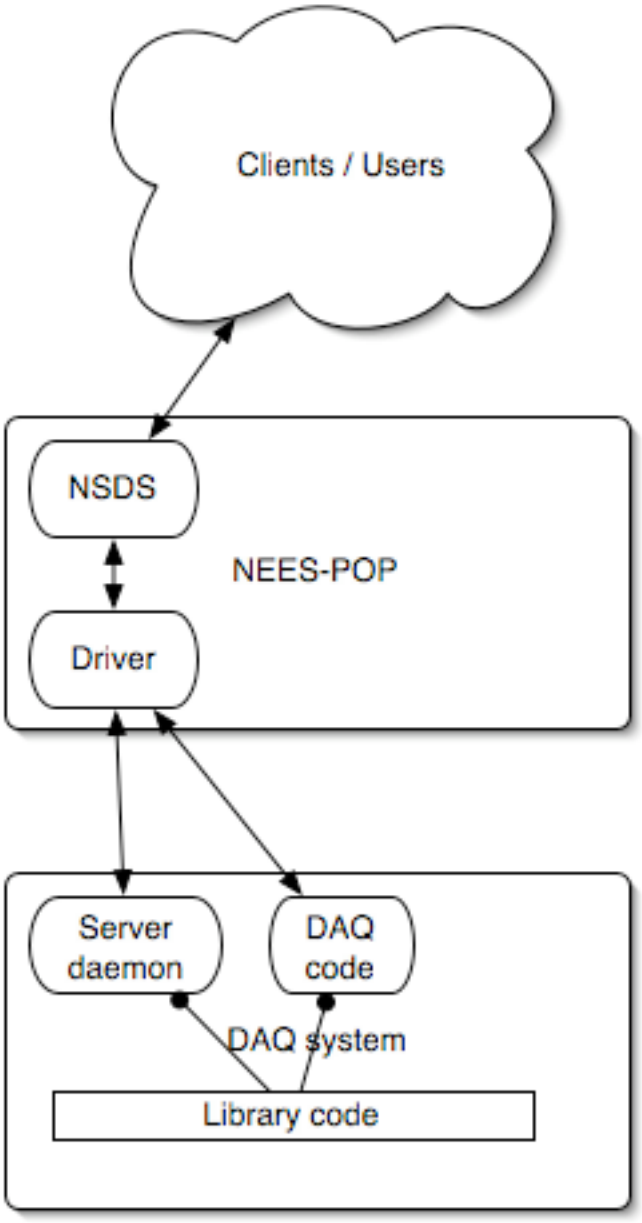
Driver means the NSDS driver. This is a piece of multithreaded C code meant to abstract the NSDS from the specifics of each different DAQ system. More on this later.

Code Access, Mailing List and Bugzilla Archive

CVS access to the NEESgrid code archive is documented at http://www.mcs.anl.gov/neesgrid/cvs.html. A Bugzilla archive for logging and tracking bugs and feature requests is available at http://bugzilla.mcs.anl.gov/neesgrid/. There is also a mailing list; instructions can be found at http://www.mcs.anl.gov/neesgrid/

Further documentation is found in nsds-driver package; there is extensive Doxygen-based documentation of the code and its workings in HTML. A copy of this documentation is mirrored at http://www.mcs.anl.gov/neesgrid/driver/index.html

Rough Design Overview



Depending how your site is configured, you may need to rewrite the driver, DAQ code, server daemon or library code. The rest of the document should clarify which portions need reworking.

Communications Overview

The main goals of this system were simplicity, portability and transparency. From that, the following choices were made:

1. All data (numeric) and commands will be sent in ASCII
2. We will assume that the network between the DAQ and driver is secure

3. We will use two TCP ports per logical connection, in the interests of keeping data and control distinct.
4. We will add a driver, so that sites not using LabVIEW can use their DAQ system with minimal effort.
5. DAQ should not know or care if the network is present, missing or unreliable. Data comes first; we stream it out if possible but strive not to interfere with DAQ.

More design information can be found in the NEESgrid System Architecture white paper.

Connections and Ports

The communications used between NSDS and driver and DAQ utilize two TCP ports. Currently, the NSDS listens on 42420/42421 and the DAQ listens on 55055/55056. However, these are changeable – the driver via the command line, and in the example LabVIEW code, it's a front panel control in the 'Server Daemon' program.

Timestamps

It is of note that this code uses ISO 8601 timestamps for all time markers. We use the UTC encoding, with fractional seconds for subsecond-sampled data. See nsds-util.c for C code to generate these; there is LabVIEW code in the subroutines library as well.

The Role of the Driver

In the provided code, the driver is a simple set of TCP pipes. Other than watching for the initial welcome message from the NSDS, all commands, data and responses are forwarded as-is. Readers may wonder *why* the driver is in the loop at all, given its lack of functionality. It's actually there only so that you can modify it for your own DAQ system. Because of that, most of the driver code is error handling and comments.

Data Channel Protocol and Format

This is as simple as possible. The format is columnar ASCII, one line per timestamp, with a newline to terminate the record. Each channel is appended as

Channel name TAB value

For example:

2002-11-13T15:48:55.26499 ATL1 -0.0064090 ATT1 -0.0042720

is a two-channel datum.

That's all there is to it, really. If your data is not simply converted to ASCII (e.g., image data), you will need to work out a data format with the NEESgrid data repository team at NCSA.

Note that we repeatedly send the channel names because the contents of the data stream will change over time as channels are subscribed or released.

Data File Format

The data file has a very similar format, with a couple of changes. It's also tab-delimited ASCII, one line per timestamp, but there's a 'metadata header' as defined by the metadata folks from NCSA, and there's no need to replicate the channel names, since the format is fixed. Here's an example to illustrate the format:

Event ID: 1757740136.322000000.0
Active channels: ATL1,ATT1,ATL3,ATT3
Sample rate: 200.000000
Channel units: g,g,in,kip

Time	ATL1	ATT1	ATL3	ATT3
2002-11-13T15:48:55.26499	-0.006409	0.004272	-0.008850	-0.007935
2002-11-13T15:48:55.41499	-0.005798	-0.003662	-0.009766	-0.006714

Note the addition of a column header line, that serves as a key to the data, and the metadata header, defining what channels are present and their engineering units. The event ID is generated and tracked by the metadata editor and is outside the scope of this document; we save and propagate it as opaque text.

Control Channel Protocol and Format

Similarly, the control channel is ASCII, one command or response per line, and newline delimited. All command are synchronous, in that the command is completed before a response is returned.

Command listing:

daq-status	Error, Offline, Unknown or Running
list-channels	Active channel listing, comma delimited.
open-port{channel name}	See below for exact syntax
close-port{channel name}	Stop subscription to given channel; see below
daq-stop	Unimplemented command
daq-start	Ditto

Sample exchange:

NSDS->driver->DAQ:

'daq-status'

DAQ->driver->NSDS:

‘Running’

NSDS->driver->DAQ:

‘list-channels’

DAQ->driver->NSDS:

‘ATL1,ATT1,ATV1,ATV2,Temp,RH’

NSDS->driver-DAQ:

‘open-port ATT1’

DAQ->driver->NSDS:

‘Streaming data on data channel from port ATT1’

(The data for sensor ATT1 will be added to the data stream on the data channel, usually by the next data point)

NSDS->driver-DAQ:

‘close-port ATT1’

DAQ->driver->NSDS:

‘Stopping data on data channel from port ATT1’

The Role of the Server Daemon

This is a standalone LabVIEW program that is running in the background It’s job is to set up the TCP connections, handle the control channel, and set global variables for the library code. For example, when a subscribe (‘open-port’) request comes in, the daemon adds the channel name to a global list of subscribed channels. The next time the DAQ sends data out, the library code will note the new subscription and add it to the outgoing datum.

Note that the server daemon and/or TCP connections are intentionally independent of the DAQ code. The global variable to mediate may seem peculiar, but it allows the daemon and DAQ to decouple nicely. DAQ can run whether or not the server is up, or if the network fails.

Sites replacing the server daemon should have little problem re-writing it in C/C++, Java or Perl – it's quite simple.

The Role of the Library Code

The library code has supporting routines for the server daemon and the DAQ code. There are subroutines for converting data into ASCII, saving to disk, and streaming via TCP. Most of this is not of interest unless you want to see how these are done with LabVIEW.

Two routines are of special interest 'Data array to NSDS stream' and 'Data array to datafile stream.' They perform similar function but have one key difference: Both take a vector of real numbers and a channel listing, but differ in their resulting ASCII. The NSDS version looks up each channel in the global list of subscribed channels, and only adds the sensor if it is needed. The datafile version always saves every sensor; the data file always contains all of the data captured.

Post-experiment FTP uploads

There are two methods of remote data access, streaming and batch. So far, we've looked at the mechanisms for delivering streaming data. However, if you look at Zombie DAQ, you will find the code to do post-experiment data uploads. We use the National Instruments Internet Toolkit, specifically the FTP transfer routines. Post-experiment, we FTP the data file to the NEES-POP or other specified destination. Once that's complete, we write a semaphore file named {datafile}.written. This triggers its transfer into the metadata handling system.

Metadata Downloading and Remote-Controlled DAQ

Mirroring the FTP uploads of data, this is a method to allow remotely controlled DAQ. The Zombie DAQ code downloads a metadata file from the repository that contains all of the information in the data file header: Channels, units, sample rate and Event ID. This is saved to metadata.ini for later use (Server daemon reads this file, for example) and DAQ begins. Note that ending condition is a front panel 'Stop' button but this could be any appropriate local condition that signals end-of-run.

The Zombie just fetches the file, and then calls the routines to parse it. If you need a different metadata format, you can work one out with the NCSA metadata team.

Testing, Benchmarking and Profiling

Once you have assimilated the design and replaced portions of it, you will need to test your components. There are several programs in the archive to assist with this.

a) fake_daq.c A simplest-case data source, useful for testing the current driver and NSDS. Useful as an end-to-end test, and possibly for benchmarking. You can set

the sample rate from the command line, and the number of simulated channels is a compile-time constant.

b) NSDS Simulator.vi LabVIEW code to emulate a normal set of operations – query DAQ status, list channels, subscribe to a requested channel, and plot the streaming data as it arrives. A good test of end-to-end functionality, it also has the side benefit of plotting the data, which is often very telling.
c) NSDS Stress Tester.vi LabVIEW code that is created for benchmarking and stress testing. It subscribes to all listed channels, which is useful.
d) Stress test Fake DAQ.vi Generates as many channels of simulated data as you request.

Note that the Server Daemon and NSDS simulator programs display the commands and responses on their front panels; this is useful for checking and response command syntax.

Unfortunately, the NSDS and Chef portions of the system do not yet have decomposition and testing tools made available. Until those are present, I recommend you use the above list to test your new components.

Profiling can take several forms: network utilization, CPU usage, sampling rate, etc. Different tools are appropriate for each; here are some I've found useful so far.

a) LabVIEW has an excellent profiler (Menu is Tools/Advanced/Profile VIs) that is quite useful in locating bottlenecks.
b) If you have Windows 2k, NT or XP, the Task Manager is useful for checking CPU, memory and network utilization.
c) On the NEES-POP, standard Unix tools such as top and lsof are invaluable for monitoring the driver and NSDS.

Where to Begin?

Having read this far, you are probably a bit adrift in new acronyms and design flotsam. Check out the nsds-driver package from CVS, and browse through fake_daq.c. It implements much of what your system will need, and is extensively commented. This code is documented in the html subdirectory, with auto-generated Doxygen pages.

Note that you will also need to check out and compile the 'flog' message library before you can compile fake_daq.

Best of luck!